**Support de Cours : Algorithmique II**

**Filière : SMI-S3 (2014-2020)**

**Prof. H. Mraoui**

**Université Mohammed Premier**
**Faculté des Sciences**
**Département d'Informatique**
**Oujda - MAROC**

**Avertissement** : Ces notes sont régulièrement mises à jour et corrigées, ne vous étonnez pas si vous découvrez des erreurs. Merci de me les communiquer.

# Table des matières


I Rappel : Algorithmique . . . . . 4
- I.1 Qu'est ce qu'un algorithme . . . . . 4
- I.2 Variables . . . . . 5
- I.3 Saisie et affichage . . . . . 6
- I.4 Affectation . . . . . 7
- I.5 Opérateurs . . . . . 7
- I.6 Tests et conditions . . . . . 8

II Fonctions et procédures . . . . . 13
- II.1 Fonctions . . . . . 14
- II.2 Procédures . . . . . 24

III La récursivité . . . . . 27
- III.1 Définitions et Exemples . . . . . 27
- III.2 Récursivité terminale . . . . . 34

IV Enregistrements et fichiers . . . . . 36
- IV.1 Structures et enregistrements . . . . . 36
- IV.2 Fichiers . . . . . 43

V La complexité . . . . . 50
- V.1 Classes de complexités classiques . . . . . 51
- V.2 Calcul de la complexité . . . . . 52
- V.3 Exemples importants . . . . . 54

VI Preuves d'algorithmes . . . . . 57
- VI.1 Propriétés recherchées . . . . . 57
- VI.2 Terminaison d'un algorithme . . . . . 58
- VI.3 Correction d'un algorithme . . . . . 58
- VI.4 Triplet de Hoare . . . . . 59
- VI.5 Preuve par invariant de boucle . . . . . 60

VI.6 Correction d'une fonctions récursive . . . . . 61

Pour la bibliographie, on pourra consulter les ouvrages suivants :

1) S., Rohaut, Algorithmique : Techniques fondamentales de programmation (avec des exemples en PHP). Editions ENI, 2009.

2) P. Wolper. Introduction à la calculabilité. Dunod, 2006.

3) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction à l'algorithmique. Dunod, 2002 (seconde édition).

# I Rappel : Algorithmique

## I.1 Qu'est ce qu'un algorithme

Le terme algorithme est employé en informatique pour décrire une méthode de résolution de problème programmable sur machine.

Un algorithme est une suite finie d'instructions à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données.

Un algorithme décrit ce qui doit faire l'ordinateur pour arriver un but bien précis. Ce sont les instructions qu'on doit lui donner. Donc l'algorithme est un moyen pour le programmeur de présenter son approche d'un problème donné à d'autres personnes, dans un langage clair et compréhensible par l'être humain.

Les opérations de base pouvant composer un algorithme sont décrites en pseudo-langage. Il s'agit d'un langage informel proche du langage naturel et indépendant de tout langage de programmation. Ce pseudo-langage sera ensuite traduit et codé dans le langage de programmation désiré.

Les étapes de conception d'un algorithme :

- Comprendre le problème.
- Identifier les données du départ (entrées) et celle(s) qu'il faut obtenir (sorties).
- Structurer les données (variables ou constantes, type...).
- Déterminer les transformations nécessaires à faire pour obtenir les résultats (traitements/développements).
- Présenter les résultats.

Un bon algorithme doit être :

- Lisibles (Compréhensible).
- De haut niveau (être traduit en un langage).
- Précis (Pas de confusion).
- Concis (ne doit pas dépasser une page).
- Structuré (Parties facilement identifiables).

La structure à utiliser pour réaliser un algorithme se présente ainsi :

| Algorithme Nom_de_algorithme<br>Variable /* Déclaration des variables */ |
| --- |
| 1. Début<br>2. Corps de l'algorithme<br>3. Fin |

## I.2 Variables

Les données manipulées dans un algorithme sont appelées des variables. Pour exister, une variable doit être déclarée, c'est-à-dire que vous devez indiquer au début de l'algorithme comment elle s'appelle et ce qu'elle doit contenir. Les variables se déclarent au début de l'algorithme, avant le programme lui-même mais après le mot "Variable".

```
Variable        /* Déclariation des variables */
        variable1 : type
        variable2,variable3,... : type
```

Donc, chaque variable est définie par :

- Un nom qui le désigne et le distingue des autres variables.
- Un type qui caractérise la nature des informations qui seront représentées par la variable et les opérations qui seront autorisées sur cette variable.
- Une valeur qui représente l'état actuel de l'objet.

Nous avons plusieurs types :

- Les constantes : désignent des références à des valeurs invariantes dans le programme.
- Les entiers : nombres sans virgule, négatifs ou positifs.
- Les réels : nombres à virgule, positifs ou négatifs.
- Les booléens : Pour déterminer si une affirmation vraie ou fausse.
- Les caractères : pour représenter un seul caractère.
- Les chaînes : ce sont une suite de caractères.
- Les tableaux : permettent de représenter un ensemble de valeurs appartenant toutes au même type.

Syntaxe de la déclaration :

Constante Nom_Constante ⟵ Valeur

Variable variable1,variable2,... : Entier

Variable variable1,variable2,... : Réel

Variable variable1,variable2,... : Caractère

Variable variable1,variable2,... : Chaîne de caractères

Variable Tab1,... : tableau[0..nbelement-1] d'entiers

Variable Tab1,... : : tableau[0..dim1-1] [0..dim2-1] de réels

## I.3 Saisie et affichage

Pour l'affichage des données, on utilise le mot "Afficher".

Pour inviter un utilisateur à rentrer au clavier une valeur, utiliser le mot "Lire".

**Exemple** :

```
Algorithme Lire
Variable x : Entier
Début
   Afficher("Saisir un entier x ")
   Lire (x)
   Afficher (" x vaut :", x )
Fin
```

## I.4 Affectation

Pour affecter une valeur à une variable, on écrit :

$$\boxed{\text{var} \longleftarrow \text{Valeur}}$$

L'affectation, notée par le symbole $\longleftarrow$, est l'opération qui évalue une expression (constante ou une expression arithmétique ou logique) et attribue la valeur obtenue à une variable.

## I.5 Opérateurs

Les opérateurs permettent de produire un effet sur les objets de l'algorithme (variables ou constante) en effectuant des calculs. des vérification d'égalités et d'inégalités. Ils sont divisés en trois classe :

Les opérateurs arithmétiques

| Opérateur | Signification | Syntaxe |
|---|---|---|
| $+$ | addition | $x + y$ |
| - | soustraction | $x - y$ |
| $*$ | multiplication | $x * y$ |
| $/$ | division | $x/y$ |
| $mod$ | modulo | $x\ mod\ y$ |
| $div$ | Division Euclidienne | $x\ div\ y$ |

**Exemple I.1**

$$\begin{aligned}
(15\ mod\ 4) &= 3 = \text{ le reste de la division euclidienne de 15 par 4;} \\
(10\ div\ 2) &= 5 = \text{ le quotient de la division euclidienne de 10 par 2;} \\
(11\ div\ 2) &= 5
\end{aligned}$$

Les opérateurs de comparaison

| Opérateur | Signification | Syntaxe |
|---|---|---|
| $=$ | égal | $x = y$ |
| $<>$ | différent | $x <> y$ |
| $<$ | inférieur | $x < y$ |
| $>$ | supérieur | $x > y$ |
| $<=$ | inférieur ou égal | $x <= y$ |
| $>=$ | supérieur ou égal | $x >= y$ |

Les opérateurs logiques

| Opérateur | Signification | Syntaxe |
|---|---|---|
| et | intersection | $x$ et $y$ |
| ou | réunion | $x$ ou $y$ |
| non | complémentaire | non y |

## I.6 Tests et conditions

**L'instruction Si**

L'instruction de test ou instruction **Si** permet de ne faire exécuter certaines instructions que si une certaine condition est remplie. Voici la syntaxe de l'instruction **Si**.

```
Si Booléen alors
   Bloc d'instructions
{Sinon
   Bloc d'instructions}    ⟵ Option Facultative
FinSi
```

**Exemples** :

```
Algorithme Structure_Alrenative_1
Variable x : Entier
Début
   Afficher("Saisir un entier x ")
   Lire (x)
   Si (x > 0) Alors
      Afficher ( x,"est un nombre positif")
   FinSi
Fin
```

```
Algorithme Structure_Alrenative_2
Variable x : Entier
Début
   Afficher("Saisir un entier x ")
   Lire (x)
   Si (x > 0) Alors
      Afficher ( x,"est un nombre positif")
   Sinon
      Afficher ( x,"est un nombre négatif ou nul")
   FinSi
Fin
```

```
Algorithme Maximum
Variable a ,b, max : Entier
Début
   Afficher ("Saisir deux entiers a et b ")
   Lire (a,b)
   Si (a > b) Alors
      max ⟵ a
   Sinon
      max ⟵ b
   FinSi
   Afficher ("le maximum de " , a , " et de " , b, " est : " , max)
Fin
```

## Choix multiples

**Suivant Cas** est une instruction spécifique pour le cas où il faudrait choisir entre plusieurs possibilités. Voici sa syntaxe.

```
Suivant Cas (expression_entière) Faire
        Cas cste1 : instructions ;
        Cas cste2 : instructions ;
        ........
        Cas csteN : instructions ;
```

Autre : instructions ;

Fin Suivant

L'expression ne peut être qu'entière. L'expression est évaluée, puis on passe directement au "Cas" correspondant à la valeur trouvée. Le cas default est facultatif, mais si il est prévu il doit être le dernier cas.

Lorsque la valeur de l'expression située après le **Suivant Cas** coincide avec l'une des constante **Cas**, le programme ne se contente pas d'exécuter la ou les instructions de la branche **Cas** concernée, mais exécute aussi toutes les instructions suivantes, cela jusqu'à la fin du **Suivant Cas**. Normalement, on ne veut pas que soient exécutées les instructions des autres branches suivantes. Donc, il faut une instruction spéciale pour assurer que seules les instructions souhaitées seront effectuées. cette instruction est l'instruction **Sortir**.

**L'instruction Pour :**

Une boucle permet de répéter plusieurs fois un bloc d'instructions.

La structure **Pour** est une boucle qui teste une condition avant d'exécuter les instructions qui en dépendent. Les instructions sont exécutées (répétées) tant que la condition est remplie (Vraie). Voici la syntaxe de l'instruction **Pour**.

```
<Initialisation>
Pour expr_initiale allant de valeur1 à valeur2 faire
   Bloc d'instructions
FinPour
```

Cette boucle est surtout utilisée lorsque l'on connaît à l'avance le nombre d'itérations à effectuer. L'*expr_initiale* est effectuée une fois, en premier. Puis on teste la condition. On effectue l'instruction puis l'incrémentation tant que la condition est vraie.

```
Algorithme Factoriel
Variable
   N : Entier
   i : Entier
   Fact : Entier
Début
   Afficher (" Saisir une valeur entière N > 0 : ")
   Lire (N)
      Fact ⟵ 1
      Pour i allant de 1 à N Faire
         Fact ⟵ Fact*i
      FinPour
   Afficher ("Le factoriel de ", N , " est : " , Fact)
Fin
```

**L'instruction Tant que :**

La boucle *Tant que* permet de faire répéter l'exécution d'instructions tant qu'une certaine condition est remplie (Vraie). Avec la syntaxe :

```
<Initialisation>
Tant que Condition Faire
   Bloc d'instructions
   Bloc de contrôle
Fin Tant que
```

```
Algorithme ExempleTantque
Variable i : Entier
Début
   i ⟵ 0
   Tant que (i < 4) Faire
      Afficher(" Bonjour tout le monde ")
      i ⟵ i + 1
   Fin Tant que
Fin
```

**L'instruction Faire jusqu'à :**

Contrairement aux structures **Pour** et **Tant que**, la boucle **Faire jusqu'à** teste sa condition après exécution de l'instruction du corps de la boucle.

```
<Initialisation>
Faire
   Bloc d'instructions
Jusqu'à Condition
```

```
Algorithme Somme
Variable
   N, S, i : Entier Début
   Afficher ("Saisir une valeur entière positive :")
   Lire (N)
   S ⟵ 0
   i ⟵ 0
      Faire
         i ⟵ i + 1
         S ⟵ S + i
      jusqu'à (i > =N)
   Afficher ("La somme : S = ", S)
Fin
```

# II Fonctions et procédures

L'objectif de ce nouveau chapitre est de vous apprendre à organiser votre code en modularisant à l'aide de ce qu'on appelle des fonctions ou procédures. les fonctions font partie des aspects de traitement au même titre que les expressions, les opérateurs et les structures de contrôle, que nous avons vues précédemment. En règle générale, les fonctions opèrent sur des données, et en retour les données vont influencer nos fonctions. Jusqu'à maintenant les algorithmes que vous avez écrits étaient constitués d'une séquence linéaire d'instructions sans organisation plus globale et sans partages des tâches répétées. Par exemple, si la tâche consistant à calculer le maximum de trois nombres comme le fait le code suivant devait être exécuté plusieurs fois dans notre algorithme. Par exemple, calculer le maximum de trois nombres à plusieurs endroits dans un plus gros algorithme. Que feriez-vous ? Vous pourriez être tenté de recopier le code autant de fois que nécessaire aux endroits appropriés, mais évidemment ceci est très mauvaise solution. Il ne faut jamais dupliquer de code en programmant. Ne soyez pas tenté par le copier-coller. Ce que vous voudriez recopier doit être mis dans une fonction. Pourquoi ne jamais dupliquer du code (copier/coller). Cela rend l'algorithme

- Inutilement long
- Difficile à comprendre
- Difficile à maintenir : il faudrait reporter chaque modification dans chacune de copies.

C'est pour ça tout bon langage de programmation fournit donc des moyens pour permettre la réutilisation de portions de programme. En algorithmique, ça se fait en utilisant des fonctions ou des procédures.

```
Si (a > b) alors
   max2 ⟵ a
Sinon
   max2 ⟵ b
FinSi
Si (c > max2) alors
   max3 ⟵ c
Sinon
   max3 ⟵ max2
FinSi
```

**Problème :**

Dès qu'on commence à écrire des programmes, il devient difficile d'avoir une vision globale sur son fonctionnement :

- Difficulté de trouver des erreurs.
- Redondance.
- Maintenance.

**Solution :**

Lorsqu'un programme est très long, il n'est pas réaliste de tout programmer d'un seul tenant. Le programme est décomposé en de plus petites parties (sous-algorithmes) qui sont ensuite appelées au moment opportun par l'algorithme principal.

Un programme complet est souvent forme une application. Une application est composée de plusieurs parties fonctionnelles :

- L'algorithme principal qui correspond du bloc principal d'instructions situées sous le mot-clé Algorithme et Début.
- Des sous-algorithmes chargés de divers rôles. C'est l'algorithme principal qui se charge souvent d'appeler des sous-algorithmes.

## II.1 Fonctions

### Déclaration et définition

**Définition II.1** *Une fonction est un sous programme particulier, accomplissant une tâche particulière et qui ne renvoie, par exemple dans l'algorithme principal, qu'un et un seul résultat.*

Pourquoi on l'utilise :

- Décomposer l'algorithme en de parties appelées par le programme principal ou un sous programme.
- Éviter la répétition inutile les mêmes instructions plusieurs fois.

Une fonction est un sous-programme qui :

- a un nom.
- peut avoir des paramètres ou arguments.
- retourne une valeur d'un certain type.
- peut avoir besoin de variables.
- est composée d'instructions.

**Remarque II.1** *Une fonction peut ne pas avoir de paramètres.*

La fonction peut être déclarée de la manière suivante :

| Fonction $Nom_de_fonction$(Liste des paramètres) : $type_de_retour$<br>Variable /∗ Déclaration des variables ∗/ |
|---|
| – DébutFonction<br>– Liste des instructions de la fonction<br>– Retourne (Résultat)<br>– FinFonction |

On utilise le mot clef "Retourne" pour dire que la fonction doit fournir le résultat en retour.

Une fonction a trois facettes. Tout d'abord son prototype, qu'on peut voir comme un résumé de ce que doit faire la fonction. Puisqu'il contient son nom, ses paramètres qui correspondent aux valeurs dont a besoin la fonction pour pouvoir fonctionner. Ainsi, que le type de la valeur que va fournir la fonction. Un deuxième aspect, est la création effective de la fonction ou sa définition qui commence comme un prototype mais contient également de ce qu'on appelle le corps de la fonction, qui est le code qui va être exécuté quand on utilise la fonction.

le dernier aspect est donc l'utilisation de la fonction ou son appel. on va utiliser la fonction en lui donnant des valeurs effectives pour ces paramètres la fonction va fournir une valeur qu'on va pouvoir affecter à une variable. En pratique le programmeur-concepteur c'est dire la personne qui va écrire la définition de la fonction n'est pas forcément la même personne que le programmeur-utilisateur, c'est dire la personne qui va utiliser la fonction. le programmeur-utilisateur n'a pas à connaître le corps de la fonction . Tout ce dont il a besoin de connaître c'est le prototype de la fonction, pour pouvoir l'appeler. Le prototype sert donc d'accord entre le programmeur-utilisateur et programmeur-concepteur. Le programmeur-utilisateur doit respecter le prototype de la fonction quand il appelle la fonction. Le programmeur-concepteur va s'arranger pour que le prototype corresponde au problème que le programmeur-utilisateur veut résoudre.

Donc, une fonction est une portion de programme réutilisable ou importante en soi. Elle est caractérisée par

- un corps : la portion d'algorithme à réutiliser ou mettre en evidence ;
- un nom : par lequel on désignera cette fonction et qui permet de faire référence à cette fonction
- des paramètres : ensembles de variables extérieures à la fonction dont le corps dépend pour fonctionne ;
- un type et une valeur de retour : ce que la fonction renvoie au reste de l'algorithme

L'utilisation de la fonction dans une autre partie de l'algorithme se nomme un appel de la fonction.

### Appel d'une fonction

Un sous-algorithme est exécuté depuis l'algorithme principal ou un autre sous-algorithme. Pour cela, le programme fait appel au sous-programme. Donc, l'appel d'une fonction : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction ou de l'algorithme principal. Plus précisément, l'appel au sous-programme est une instruction qui va déclencher l'exécution de celui-ci. Ainsi, l'appel va fournir le résultat de l'exécution de la fonction. Ce résultat peut, ensuite, par exemple, être stocké dans une variable.

Voici deux exemples d'appel :

var ⟵ Nom_de_fonction(var1, var2,..)

Afficher (Nom_de_fonction(var1, var2,..))

**Exemples :**

Fonction qui renvoie le maximum de deux variables réelles :

```
Fonction Maximum2 (x : Réel, y : Réel) : Réel
Variable
  max2 : Réel
DébutFonction
    Si (x > y) alors
      max2 ⟵ x
    Sinon
      max2 ⟵ y
    FinSi
    Retourne max2
FinFonction
```

Fonction qui renvoie le maximum de trois variables réelles :

```
Fonction Maximum3 (x : Réel, y : Réel, z : Réel) : Réel
Variable
  max3 : Réel
DébutFonction
      max3 ⟵ Maximum2(Maximum2(x ,y) ,z)
    Retourne max3
FinFonction
```

Programme principal qui calcule le maximum de trois variables.

```
Algorithme Maximum_de_trois_variables
Variable
   a,b, c,max3 : Réel
Début
   Afficher (" Saisir 3 nombre réels : ")
   Lire (a,b,c)
   max3 ⟵ Maximum3(a,b,c)
   Afficher ("Le maximum est = ", max3)
Fin
```

**Evaluation d'un appel de fonction**

Nous allons maintenant nous intéresser à ce que se passe lorsqu'on utilise, lorsqu'on appelle une fonction. Plus précisément, nous allons détailler ce qui se passe au moment où on appelle une fonction. Reprenons l'exemple du calcul le maximum de trois nombres. Nous avons ici le cas d'un petit algorithme principal qui demande à l'utilisateur d'introduire trois nombres, qui lit ces trois nombres depuis l'entrée standard et qui affiche le maximum de ces trois nombres. Le calcul du maximum est réalisé au moyen d'un appel de fonction, et c'est les mécanismes appliqués par cet appel de fonction que nous allons étudier.

Plaçons-nous dans une situation concrète. Que se passe-t-il lors de l'appel suivant

```
max3 ⟵ Maximum3(1.5+0.8, 3.4*1.25, 5)
```

Les arguments passés à la fonction "Maximum3" au moment de l'appel correspondent évidemment aux paramètres attendus par la fonction pour qu'elle puisse s'exécuter.
Notez que l'on appel usuellement "paramètres" les données nécessaires à la fonction pour qu'elle puisse s'exécuter telle décrite dans le prototype, et par "argument" les valeurs que l'on passe effectivement à la fonction au moment où on l'invoque, où on l'appelle. Donc nous allons maintenant disséquer les différentes étapes qui ont lieu lorsqu'un tel appel est réalisé. Première étape, on commence par évaluer les arguments passés à la fonction "Maximum3" au moment de l'appel. Seconde étape, il s'agit maintenant d'apparier les arguments d'appel de la fonction avec

les paramètres attendus par la fonction "Maximum3". Cet appariement se fait par le biais d'une affectation, donc on va affecter à $x$ le résultat de l'évaluation du premier argument, à savoir 2.3, et on va affecter à $y$ le résultat de l'évaluation du second argument, à savoir 4.25, et finalement on va affecter à $z$ la valeur 5. La fonction "Maximum3" dispose désormais de valeurs concrètes dans $x$, $y$ et $z$, avec lesquelles elle va pouvoir travailler, s'exécuter. Troisième étape, on commence l'exécution de la fonction "Maximum3", ici en l'occurrence, toutes les instructions précédant cette instruction terminale, l'instruction "Retourne" vont être exécutées. Enfin, l'instruction "Retourne" est exécutée. Par fois, On commence par évaluer l'expression suivant le mot clé "Retourne". Ultime étape, on met la valeur produite par l'expression, on met cette valeur à disposition de celui qui a invoqué (appelé) la fonction "Maximum3" par le biais d'une instruction "Retourne", ce qui veut dire que cet appel de fonction peut désormais être remplacé par la valeur résultant de l'appel de fonction, à savoir 5. Donc, à l'issue de la dernière étape la variable "max3" ôte ici la valeur 5

Voici donc un petit résumé des différentes étapes que nous venons d'examiner. Lorsqu'un appel de fonction a lieu dans un algorithme, cinq étapes ont lieu

1. Les expressions passées en argument sont évaluées.
2. Les valeurs résultant de l'évaluation de ces expressions sont affectées aux paramètres de la fonction.
3. Les paramètres de la fonction disposent désormais de valeurs concrètes avec lesquelles la fonction peut travailler, le corps de la fonction s'exécute.
4. L'expression qui suit la première commande "Retourne" rencontrée à l'exécution est évaluée.
5. Le résultat de cette évaluation est retourné comme résultat de l'appel, ce qui en clair signifie que cette valeur remplace désormais l'expression de l'appel.

Ceci est ce qui se produit dans le cas le plus général. Le cas où une fonction a besoin, pour travailler, de données entrantes d'arguments, de paramètres entrants, et fournit en sortie une valeur concrète. Il s'agit du cas le plus général. Il existe des situations où ce schéma en cinq étapes est un petit simplifié. Première situation, il existe des cas où une fonction peut par exemple fournir un résultat en sortie, mais n'a pas besoin de données entrantes pour pouvoir travailler. Donc, dans ce cas de figure, puisqu'il n'y a pas besoin d'arguments en entrée, les étapes 1 et 2

n'ont pas lieu. Deuxième cas de figure, il peut se produire des situations où une fonction réalise des traitement, mais ne fournit en sortie aucune valeur concrète. Par exemple, une fonction peut réaliser un certain nombre d'affichages sur un terminal, mais ne pas fournir de valeur en sortie et dans ce cas de figure, les étapes 4 et 5 n'ont pas lieu. Il existe un passage d'arguments particulier, que l'on examine dans la programmation par $C++$, qui s'appelle "le passage par référence", où l'étape n'a pas lieu.

Considérons une fonction définie comme suit

```
Fonction f(x : Entier) : Entier
Variable
   ....
DébutFonction
      ....          Retourne ...
FinFonction
```

Et voici enfin une petite synthèse de l'appel fonctionnel.

$$\boxed{z \longleftarrow f(y)}$$

Donc, une fonction peut être appelée dans un programme. Au moment de l'appel, il y a un appariement qui est fait entre les arguments d'appel et les paramètres de la fonction. Ici, concrètement, l'argument $y$ va être copié dans le paramètre $x$. Le paramètre $x$ a désormais une valeur concrète avec laquelle le corps de la fonction peut s'exécuter. Il faut noter que les deux cases mémoires associées aux deux variables $x$ et $y$ sont distinctes. L'expression, qui suit le "Retourne" est évaluée et mise à la disposition de l'algorithme appelant par le biais de l'instruction "Retourne". Donc, l'appel de la fonction est remplacé par le résultat de l'évaluation du "Retourne" et peut être affecté à une variable du programme appelant.

**Remarque II.2** *Lorsque on dit qu'une valeur est passée en argument d'une fonction, on signifie que cette valeur est copiée dans un paramètre de la fonction. Lorsque on dit qu'une fonction retourne une valeur $y$, on signifie que l'expression de l'appel de la fonction est remplacée par la valeur retournée.*

En résumé, on doit d'abord identifié le code qui devait être mis dans une fonction. On extrait ce code pour pouvoir créer une nouvelle fonction que on a nommée. On a identifié ce dont avait besoin en entrée notre fonction pour pouvoir fonctionner. On a aussi identifié ce qui devait fournir notre fonction au reste de l'algorithme. Maintenant, on peut remplacer la partie du code que nous avons extraite, par un appel à notre fonction.

### Variables locales et globales

L'endroit où les variables sont déclarées est très important. Selon cet endroit les variables ont une portée différente. La portée d'une variable est sa visibilité au sein des différentes parties du programme.

**Définition II.2** *Les variables accessibles uniquement par le programme ou sous-programme dans lesquels elles sont déclarées, sont appelées des variables locales.*

Les variables locales de même nom n'ont aucun rapport entre elles. Elles sont totalement indépendantes les unes des autres et aucune interaction entre elles n'est possible. Donc, les variables locales peuvent donc parfaitement porter un même nom. Du côté de la mémoire, le contenu des variables locales, qui portent le même nom, est cloisonné et distinct, à des adresses (cases) différentes.

D'autre part, il serait pourtant très pratique de pouvoir accéder à une variable depuis n'importe quel endroit du programme, qu'il soit principal ou un sous-programme. La portée d'une telle variable s'étendrait à tout le code. Ce type de variable s'appelle une variable globale.

Les variables globales sont déclarées de cette manière (en dehors des sous-programmes et du programme) :

| Variable Globales | a : Réel |
|---|---|

### Passage des arguments

Dans la section précédente, nous avons examiné les cinq étapes qui se déroulent le plus général lorsqu'on appelle une fonction. Dans les exemples vus jusqu'à lors, les arguments passés à la fonction étaient soit de simples valeurs, soit des expressions à évaluer. Nous allons maintenant examiner un peu plus finement ce qu'il se

passe lorsque les arguments passés à la fonction sont des variables. Partons d'un exemple concret.

```
Procédure f(x : Entier)
DébutProcédure
   x ⟵ x+1
   Afficher("x= ", x)
FinProcédure
```

```
Algorithme Exemple_passage_argument
Variable val : Entier
Début
   val⟵ 1
   f(val)
   Afficher("val= ", val)
Fin
```

Nous avons ici un petit algorithme principal dont la première instruction consiste à déclarer une variable "val" et l'initialiser avec la valeur 1. La seconde instruction de cet algorithme appelle la fonction "f" qui est définie ici, en lui fournissant justement en argument la variable "val" que nous venons juste de déclarer. Nous avons vu dans la partie précédente qu'au moment de cet appel il y a une mise en correspondance entre l'argument d'appel de la fonction et la liste des paramètres attendus par la fonction. Donc, ici nous avons deux entités, nous avons une variable "x" utilisée par la fonction. Et il se trouve que la fonction "f" altère cette variable, la modifie en l'incrémentant d'une unité. la question que nous nous posons maintenant est la suivante : Est-que "x" et "val" sont la même zone mémoire ? Ce que veut dire concrètement que l'altération de "x" aura une incidence sur "val" ou non. Il existe en programmation (langage Machine) deux types de passage d'arguments. Le passage par valeur et le passage par référence. Donc examinons sur le même exemple que vu précédemment ce qui se produit dans l'un ou l'autre des cas. Donc, nous avons une petite fonction "f" qui prend en paramètre un entier "x" elle ne retourne aucune valeur, d'ailleurs il n'y a pas d'instruction "Retourne". Et ce que réalise cette fonction est d'incrémenter la valeur de son paramètre "x" égal à "x+1". Si une fonction utilise le passage par valeur, cela signifie que le paramètre

"x" de la fonction correspond à une zone locale à la fonction. Ce qui signifie que, au moment où on invoque la fonction en lui passant en argument une variable, variable qui est définie à un autre endroit du programme comme par exemple ici, dans l'algorithme principal. Au moment où on appelle la fonction, on va copier la valeur de la variable passée en argument dans la zone locale à la fonction. Ce qui signifie que lorsque on exécute ce traitement, on va altérer la zone locale, la variable passée comme argument. ceci correspond au passage par valeur. Pour indiquer que l'on souhaite utiliser un passage par référence (par adresse) on spécifie l'en-tête de la fonction d'une façon un peu particulière. Donc, on indique par un symbole particulier que la fonction "f" travaille par référence (par adresse). Lorsqu'on définit l'en-tête de la fonction de cette façon là, on indique qu'au moment de l'appel "x" le paramètre de la fonction est une référence (adresse) à la variable passée en argument, donc est simplement un autre nom pour la variable "val" passée en argument. Ici la variable "val" valait donc 1, et le nom "x" référence le même emplacement mémoire qui est le même que celui occupé par "val". Et dans le cas présent, l'incrémentation de "x" va correspondre aussi à une incrémentation de "val".

Donc, on distingue deux types de passages d'arguments

1. Lors d'un passage par valeur, la fonction va travailler sur une copie de l'argument, ce qui signifie que les modifications que l'on fait à l'intérieur de la fonction ne sont pas répercutées à l'extérieur de la fonction. On travaille sur une zone locale, et l'argument n'est pas altéré.
2. lorsqu'on utilise le passage par référence, on indique que la variable locale qui est passée en argument correspond à une référence sur l'objet associé à l'argument lors de l'appel, et donc la différence principale c'est que toute modification effectuée à l'intérieur de la fonction se répercute aussi, est visible aussi, une fois que la fonction a terminé son exécution. Le passage par référence doit être explicitement indiqué en utilisant le symbole de l'adresse après le type.

Pour résumer, lors d'un passage par valeur le paramètre de la fonction est une zone locale à la fonction et toute altération n'a d'incidence que sur la zone locale, ne se répercute pas sur la variable passée en argument. lors d'un passage par référence, on indique que le paramètre de la fonction n'est autre qu'un nom sup-

plémentaire pour la variable passée en argument et toute altération du paramètre altérera aussi la variable passée en argument. Si l'on prend maintenant notre petit exemple introductif, nous avions donc un petit programme principal qui déclarait une variable "val" donc le contenu est 1. Ensuite, il y avait un appel de fonction. Si l'on examine, l'en-tête de la fonction, il n y a aucun symbole particulier, ce qui signifie que l'on utilise ici le passage par valeur. Puisqu'on utilise le passage par valeur cela signifie que "x", le paramètre de la fonction est une zone locale et que donc la valeur de "val" est copiée à l'intérieur de "x". L'altération de "x" n'a d'incidence que sue "x". Si on affiche "x", on perçoit bien la modification de "x", par contre lorsqu'on a terminé d'exécuter la fonction et qu'on fait afficher la valeur de "val", on voit bien que la valeur d'origine reste inchangée, et donc que "val" vaut toujours "1".

#### Transmission d'un tableau en paramètre d'une fonction

Le passage d'un tableau comme paramètre d'une fonction est impraticable en tant que valeur : la recopie du tableau prendrait trop de temps et de place. On passe donc à la fonction l'adresse du tableau, ce qui permettra à la fonction d'effectuer des modification directement sur le tableau. En général, l'identificateur du tableau (son nom) représente l'adresse du début du tableau. La notation T équivaut à l'adresse de la case mémoire T[0].

Pour la transmission d'un tableau en paramètre d'une fonction, on utilise la syntaxe suivante :

```
Fonction Nom_de_fonction (nom_tableau : tableau[ ] de Type,...)
```

Pour transmettre l'adresse du tableau à une fonction, il suffit de lui donner le nom du tableau :

```
Nom_de_fonction (nom_tableau,...) /* appel de fonction avec argument tableau */
```

## II.2 Procédures

Dans certains cas, on peut avoir besoin de répéter une tâche dans plusieurs endroits, mais que dans cette tâche on ne calcule pas de résultats ou qu'on calcule

plusieurs résultats à la fois. Dans ce cas on ne peut pas utiliser une fonction, on utilise une procédure.

**Définition II.3** *Une procédure est un sous programme semblable à une fonction mais qui retourne rien.*

La procédure s'écrit en dehors de l'algorithme principal sous la forme :

```
Procédure Nom_de_procédure(Liste des paramètres )
Variable /* Déclaration des variables */
-------------------------------------------------
- DébutProcédure
- Liste des instructions de la procédure.
- FinProcédure
```

**Exemple :**

- On propose d'écrire une fonction qui renvoie l'indice de la valeur maximale d'un tableau à partir d'un rang donné.
- Écrire une procédure qui prend en argument un tableau et permet le trier par ordre décroissant.
- Écrire l'algorithme principal.

```
Fonction indice_val_max (T : tableau[ ] d'entiers, taille : Entier, k :
Entier) : Entier
Variable i, imax, max : Entier
-------------------------------------------------
DébutFonction
  max ⟵ T[k]
  imax ⟵ k
    Pour i allant de k+1 à taille-1 Faire
      Si ( T[i]>max) Alors
        max ⟵ T[i]
        imax ⟵ i
      FinSi
    FinPour
    Retourne imax
FinFonction
```

```
Procédure Tri_décroissant (T : tableau[ ] d'entiers, taille : Entier)
Variable i, imax, a,k : Entier
DébutProcédure
    Pour k allant de 0 à taille-2 Faire
      imax ⟵ indice_val_max(T,taille,k)
      a ⟵ T[k]
      T[k] ⟵ T[imax]
      T[imax] ⟵ a
    FinPour
FinProcédure
```

```
Algorithme Programme_Pricipal
Constante n ⟵ 10
Variable
    T : tableau[ 0..n-1] d'entiers
    i : Entier
Début
    Pour i allant de 0 à n-1 Faire
      Lire (T[i])
    FinPour
    Tri_décroissant (T,n)
    Pour i allant de 0 à n-1 Faire
      Afficher (T[i])
    FinPour
Fin
```

# III La récursivité

## III.1 Définitions et Exemples

**Définition III.1** *Un sous-algorithme récursif est un sous-programme qui peut s'appeler lui-même.*

Il existe deux types de récursivité :

- Directe ou simple : le sous-programme s'appelle lui-même.
- Indirecte ou croisée : deux sous-programmes s'appellent l'un l'autre : le premier appelle le second, qui appelle le premier, etc.

**Remarque III.1** *La récursivité peut être appliquée tant aux fonctions qu'aux procédures.*

Modèle de fonction récursive directe :

```
Fonction recursive ( ... ) : ....
Variable
DébutFonction
      /* instructions */
   recursive (...)
      /* instructions */
FinFonction
```

Modèle de fonction récursive indirecte :

```
Fonction recursive_A ( ... ) : ....
Variable
DébutFonction
      /* instructions */
   recursive_B (...)
      /* instructions */
FinFonction
```

```
Fonction recursive_B ( ... ) : ....
Variable
DébutFonction
      /* instructions */
   recursive_A (...)
      /* instructions */
FinFonction
```

Une fonction récursive contient un ou plusieurs paramètres qui évoluent lorsqu' on appelle la fonction jusqu'à satisfaire un test qui nous permettra de sortir de

la fonction. La récursivité solutionne un problème en résolvant le même problème mais donne une solution plus simple. Le processus de simplification se poursuit jusqu'à l'atteinte d'un cas où la solution est connue.

```
Fonction recursive ( ... ) : ....
Variable /* Déclaration des variables */
DébutFonction
        /* instructions */
    Si (test d'arrêt) Alors
        /* instructions */
    Sinon
        recursive (...)
    FinSi
        /* instructions */
FinFonction
```

**Exemple :**

Une factorielle est l'exemple classique d'application d'un algorithme récursif :

$$Fact(n) := n! = \begin{cases} 1, & \text{Si } n = 0 \\ (n-1)! * n = Fact(n-1) * n, & \text{Si } n > 0. \end{cases}$$

```
Fonction Fact ( n : Entier ) : Entier
DébutFonction
    Si (n=0) Alors
        Retourne 1
    Sinon
        Retourne n*Fact( n-1 )
    FinSi
FinFonction
```

L'algorithme principal pour calculer $n!$ d'une manière récursive est donné par :

```
Algorithme Factorielle_d'_un_nombre
Variable      n : Entier
Début
      Afficher ("Saisir un nombre entier : ")
      Lire (n)
      Afficher ("La factorielle de ", n , " est " , Fact(n))
Fin
```

**Exemple :**

Calcul de la suite de Fibonacci :

$$Fib(n) = \begin{cases} 1, & \text{Si } n = 0 \text{ ou } n = 1 \\ Fib(n-1) + Fib(n-2), & \text{Si } n > 1. \end{cases}$$

```
Fonction Fib ( n : Entier ) : Entier
DébutFonction
      Si (n=0) ou (n=1) Alors
         Retourne 1
      Sinon
         Retourne Fib(n-1)+Fib(n-2)
      FinSi
FinFonction
```

L'algorithme principal pour calculer la suite de Fibonacci :

```
Algorithme Suite_de_Fibonacci
Variable      n : Entier
Début
      Afficher ("Saisir un nombre entier : ")
      Lire (n)
      Afficher ("Fibonacci du rang ", n , " est " , Fib(n))
Fin
```

**Exemple :**

Calcul le produit des éléments d'un tableau **T** d'une manière récursive :

- Notons par **ind_initial** et **ind_final** les indices de la première case et la dernière case du tableau $T$.

- Multiplier les éléments d'un tableau de taille **n** revient à multiplier **T[ind_initial]** avec le produit du reste (tableau de taille **n**-1).
- Le produit s'arrête si on vérifie que **ind_initial**=**ind_final**.

| Fonction Produit(T : tableau[ ] d'entiers, ind_initial : Entier, ind_final : Entier) : Entier |
|---|
| DébutFonction<br>Si (ind_initial=ind_final) Alors /* Point d'arrêt */<br>retourne T[ind_initial]<br>Sinon /* Appel récursif */<br>retourne T[ind_initial]*Produit (T,ind_initial+1,ind_final)<br>FinSi<br>FinFonction |

**Exemple :** (**Recherche Dichotomique**)
On suppose qu'on dispose d'un tableau et on se donne un élément quelconque et on cherche si cet élément est dans le tableau ou non.

**Attention** : On utilise la technique de "Recherche Dichotomique" uniquement si le tableau est déjà trié.

Notons par **ind_min** et **ind_max** les indices de la première case et la dernière case du tableau $T$.
Donc, à chaque étape

- Tester si le tableau est vide (Arrêt des appels récursifs avec échec).
- Calculer l'indice moyen (indice_max+indice_min)/2.
- Comparer la valeur présente à l'indice moyen avec l'élément recherché :
  1. Si l'élément recherché **x** est à l'indice moyen (arrêt succès).
  2. Si l'élément **x** est supérieur à la valeur T[indice_moyen] relancer la recherche avec le tableau supérieur.
  3. Sinon relancer la recherche avec le tableau inférieur.

```
Fonction Recherche_Dicho(T : tableau[ ] d'entiers, ind_min : Entier,
ind_max : Entier, x : Entier) : booléen
Variable    ind_moyen : Entier
DébutFonction
    Si (ind_min>ind_max) Alors    /* Point d'arrêt */
      Retourne Faux
    FinSi
    ind_moyen ⟵ (ind_min+ind_max) div 2
    Si ( T[ind_moyen]=x) Alors
      retourne Vrai
    Sinon
      Si ( T[ind_moyen]>x) Alors
        Retourne Recherche_Dicho(T,ind_min,ind_moy-1,x)
      Sinon
        Retourne retourne Recherche_Dicho(T,ind_moy+1,ind_max,x)
      FinSi
    FinSi
FinFonction
```

```
Algorithme Appel_Fonction_Recherche_Dicho
Constante n ⟵ 10
Variable
    T : tableau[ 0..n-1] d'entiers
    i : Entier
Début
  Pour i allant de 0 jusqu'à n-1 Faire
    Lire (T[i])
  FinPour
  Afficher (" Donner un entier : " )
  Lire (x)
  Afficher ("Résultat de la recherche est " , Recherche_Dicho(T,0,n-1,x))
Fin
```

**Exemple :** (**Tri rapide**)

L'algorithme de tri rapide, "quick sort" en anglais, est un algorithme de type

dichotomique. Son principe consiste à séparer l'ensemble des éléments en deux parties.

Nous voulons donc un sous-algorithme récursif qui permet de trier un tableau. Son principe est de parcourir le tableau **T** en la divisant systématiquement en deux sous-tableaux **T1** et **T2**. L'un est tel que tous ses éléments sont inférieurs à tous ceux de l'autre tableau et en travaillant séparément sur chacun des deux sous-tableaux en réappliquant la même division à chacun des deux sous-tableaux jusqu'à obtenir uniquement des sous-tableaux à un seul élément.

Pour effectuer la séparation, une valeur pivot est choisie. Plus précisément, pour partitionner un tableau en deux sous-tableaux **T1** et **T2** :

- on choisit une valeur quelconque dans le tableau **T** (la dernière par exemple) que l'on dénomme pivot,
- puis on construit le sous-tableau **T1** comme comprenant tous les éléments de **T** dont la valeur est inférieure ou égale au pivot,
- et l'on construit le sous-tableau **T2** comme constitué de tous les éléments dont la valeur est supérieure au pivot.

Appliquons cette démarche à l'exemple suivant : T = [ 4, 23, 3, 42, 2, 12, 45, 18, 38, 15 ]. Prenons comme pivot la dernière valeur pivot=15.

Balayage à gauche :

- $4 < 15 \Rightarrow$ il est dans le bon sous-tableau, on continue,
- tableau en cours de construction : [ 4, 15],
- $23 > 15 =>$ il est mal placé il n'est pas dans le bon sous-tableau, on arrête le balayage gauche,
- tableau en cours de construction : [ 4, 23, 15].

Balayage à droite :

- $38 > 15 \Rightarrow$ il est dans le bon sous-tableau, on continue,
- tableau en cours de construction : [4, 23, 15, 38].
- $18 > 15 \Rightarrow$ il est dans le bon sous-tableau, on continue,
- liste en cours de construction : [4 ,23, 15, 18, 38],
- $45 > 15 \Rightarrow$ il est dans le bon sous-tableau, on continue,
- tableau en cours de construction : [4, 23, 15, 45, 18, 38],
- $12 < 15 \Rightarrow$ il est mal placé il n'est pas dans le bon sous-tableau, on arrête le balayage droit,

– tableau en cours de construction : [ 4, 23, 15, 12, 45, 18, 38].
Maintenant, on va échange les deux éléments mal placés 23 et 12 :

[ 4, 23, 15, 12, 45, 18, 38] $\Rightarrow$ [ 4, 12, 15, 23, 45, 18, 38].

On reprend le balayage gauche à l'endroit où l'on s'était arrêté :
– $3 < 15 \Rightarrow$ il est dans le bon sous-tableau, on continue,
– tableau en cours de construction : [4, 12, 3, 15, 23, 45, 18, 38]
– $42 > 15 \Rightarrow$ il est mal placé il n'est pas dans le bon sous-tableau, on arrête de nouveau le balayage gauche,
– tableau en cours de construction : [4, 12, 3, 42, 15, 23, 45, 18, 38],

On reprend le balayage droit à l'endroit où l'on s'était arrêté : [4, 12, 3, 42, 2, 23, 45, 18, 38, 15]
– $2 < 15 \Rightarrow$ il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
– tableau en cours de construction : [ 4, 12, 3, 42, 15, 2, 23, 45, 18, 38],

Maintenant, on procède à l'échange 42 et 2 : [ 4, 12, 3, 2, 15, 42, 23, 45, 18, 38]. Donc, nous obtenons T1 = [4, 12, 3, 2] et T2 = [42, 23, 45, 18, 38]. Ainsi, à cette étape voici l'arrangement de T :ă

$$T = T1 + \text{pivot} + T2 = [4,\ 12,\ 3,\ 2,\ 15,\ 42,\ 23,\ 45,\ 18,\ 38].$$

Ensuite, les deux ensembles T1 et T2 sont triés séparément, suivant la même méthode. Donc, le sous-algorithme de tri rapide est récursif.

| Procédure echanger(T : tableau[ ] d'entiers, a : Entier, b : Entier)<br>Variable temp : Entier |
|---|
| DébutProcédure<br>temp $\longleftarrow$ T[a]<br>T[a] $\longleftarrow$ T[b]<br>T[b] $\longleftarrow$ temp<br>FinProcédure |

```
Procédure TriRapide (T : tableau[ ] d'entiers, g : Entier, d : Entier)
Variable    i,j,pivot : Entier
DébutProcédure
  i ⟵ g
  j ⟵ d
  pivot ⟵ T[d]
    Faire
      Tant que T[i]<pivot Faire
        i ⟵ i+1
      FinTantque
      Tant que T[j]>pivot Faire
        j ⟵ j-1
      FinTantque
        Si (i<=j) Alors
          echanger(T, i, j)
          i ⟵ i+1
          j ⟵ j-1
        FinSi
    Jusqu'à (i>j)
    Si (g<j) Alors
        TriRapide(T,g,j)
    FinSi
quad    Si (i<d) Alors
        TriRapide(T,i,d)
    FinSi
FinProcédure
```

## III.2 Récursivité terminale

Pour une fonction récursive, on parlera de récursivité terminale si aucune instruction n'est exécutée après l'appel de la fonction à elle-même. Plus précisément, une définition de fonction $f$ est récursive terminale quand tout appel récursif est de la forme "Retourne $f(\ldots)$" ; autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur

cette valeur. Par exemple, l'appel récursif de la factorielle

$$\text{Retourne} \quad n * f(n-1)$$

n'est pas terminal, puisqu'il y a multiplication par $n$ avant de retourner. Par contre, l'appel récursif dans

```
Fonction f( n : Entier, k : Entier ) : Entier
DébutFonction
    Si (n=0) Alors
      Retourne k
    Sinon
      Retourne f(n-1,k)
    FinSi
FinFonction
```

est terminal ; le paramètre $k$ joue le rôle d'un accumulateur, où l'appel initial est $f(n, 1)$.

Considérons l'exemple suivant :

```
Fonction f( n : Entier, a : Entier, b : Entier) : Entier
DébutFonction
    Si (n=0) Alors
      Retourne a
    Sinon
      Retourne f(n-1,b,a+b)
    FinSi
FinFonction
```

avec l'appel initial est $f$(n,1,1). Donc, la fonction $f$ permet de transformer une définition récursive non terminale, de la suite de Fibonacci, en une récursivité terminale pour optimiser l'exécution et la complexité.

# IV Enregistrements et fichiers

## IV.1 Structures et enregistrements

### Type structuré

La faculté des sciences d'oujda organise les informations concernant la filière SMI-S3 dans une liste identique à la suivante :

| Numéro d'examen | Nom et Prénom | ... | Moyenne | Observation |
|---|---|---|---|---|
| 1 | Idrissi Hassan | ... | 12 | Validé |
| 2 | Salhi Yahia | ... | 09 | Non Validé |

**Problème** : Le chef de filière veut créer un programme permettant la saisie et le traitement de cette liste sachant qu'elle comporte au maximum 200 étudiants. Pour faire ça, on doit tout d'abord

- Donner la structure de données nécessaire pour les objets à utiliser.
- Donner une déclaration algorithmique de ces objets.

La solution naïve consiste à utiliser les objets suivants

| Objet | Type / Nature | Rôle |
|---|---|---|
| Num | Tableau de 200 chaînes | Tableau des numéros des étudiants |
| Nom | Tableau de 200 entiers | Tableau contenant les noms et prénoms |
| ... | ... | ... |
| Moy | Tableau de 200 réels | Tableau des moyennes |
| Obser | Tableau de 200 chaînes | Tableau des observations |

Malheureusement, cette approche est totalement ingérable dès qu'il s'agit de :

- trier les moyennes,
- rechercher la liste des étudiants ayant validé le semestre,
- rechercher la liste des étudiants autorisés à passer l'examen de rattrapage.

Pour proposer une solution pratique, il faudrait une sorte de type particulier qui pourrait regrouper en une seule liste des variables de types différents. Ces types existent. Ils s'appellent des structures et permettent de décrire des enregistrement. Les enregistrements sont des structures de données dont les éléments peuvent être de type différent et qui se rapportent à la même entité. Les éléments qui composent un enregistrement sont appelés champs.

- Avant de déclarer une variable structure, il faut avoir au préalable définit son type, c'est à dire le nom et le type.
- Un type structuré doit être déclaré et défini avant les variables pour qu'il puisse être utilisé pour définir des variables de type structuré.
- Vous devez déclarer les types structurés hors de l'algorithme et des sous-algorithmes.

**Définition IV.1** *Une structure est un type de données défini par l'utilisateur et qui permet de grouper un nombre fini d'éléments (ou champs) de types éventuellement différents.*

La syntaxe à utiliser pour définir un nouveau type structuré :

```
Type
Structure nom_type
champ1 : type_champ1
................
champn : type_champn
FinStructure
```

La structure est déclarée dans section particulière sous le mot-clé "**Type**", entre les mots-clés **Structure** et **FinStructure**.

- Chaque structure porte un nom. Ce nom sera utilisé pour déclarer des enregistrements.
- Une structure peut contenir 1 à $n$ champs, du même type ou de types différents.
- Une structure a un seul champ est en soi totalement inutile.

**Exemple :**

```
Type
Structure tetudiant
Num : Entier
Nom : chaîne de caractères
Moy : Réel
Oberv : chaîne de caractères
FinStructure
```

Une fois qu'on a défini un type structuré, on peut déclarer des variables enregistrements exactement de la même façon que l'on déclare des variables d'un type primitif.

**Syntaxe :**

```
Variable      nom_var : nom_type
```

**Exemple :**

```
Variable      étudiant, étudiant1, étudiant2 : tétudiant
```

Alors que les éléments d'un tableau sont accessibles au travers de leur indice, les champs d'un enregistrement sont accessibles à travers leur nom, grâce à l'opérateur '.', i.e., vous accédez aux champs d'un enregistrement en passant par le nom de l'enregistrement et le nom du champ séparé par le caractère '.' , le point, selon la forme suivant :

```
nom_enreg.nom_champ
```

Reprenons l'exemple précédent :

```
étudiant.Num ⟵ 1
étudiant.Nom ⟵ Idrissi Hassan
étudiant.Moy ⟵ 12
étudiant.Obser ⟵ Validé
```

Donc, Les champs d'un enregistrement se manipulent exactement comme des variables. Ils peuvent recevoir des valeurs et leur valeur peut être affectée à une autre variable. Les champs peuvent être utilisés partout où les variables sont utilisées, y compris comme paramètres de de sous-programmes, en saisie, en affichage, etc.

**Exemple :**

Prendre un catalogue de produits dans un magasin. Un article est décrit par une référence, un nom et un prix.

```
Type
Structure tarticle
      ref : chaîne de caractères
      nom : chaîne de caractères
      prix : Réel
FinStructure
```

```
Variable
   article1, article2 : tarticle
   réponse : chaîne de caractères
Début
   Afficher (" Référence du premier article ? ")
   Lire (article1.ref)
   Afficher (" Nom du premier article ? ")
   Lire (article1.nom)
   Afficher (" Prix du premier article ? ")
   Lire (article1.prix)
   Afficher (article1.ref, article1.nom, article1.prix)
   Afficher (" Copier le premier article dans le second ? ")
   Lire (réponse)
   Si (reponse= "oui") Alors
      article2 ⟵ article1
   FinSi
   article2.prix ⟵ 15.25
   Si (article2.prix = article1.prix) Alors
      Afficher (" Les deux articles ont le même prix ")
   FinSi
Fin
```

**Imbrication d'enregistrements**

Un type structuré peut être utilisé comme type pour des champs d'un autre type structuré.

**Exemple :**

```
Type
Structure tfabricant
   ref : chaîne de caractères
   nom : chaîne de caractères
   tel : chaîne de caractères
FinStructure
```

```
Type
Structure tarticle
      ref : chaîne de caractères
      nom : chaîne de caractères
      prix : Réel
      fab : tfabricant
FinStructure
```

Maintenant déclarez un enregistrement de type tarticle :

```
Variable        article1 : tarticle
```

Pour accéder aux champs de l'enregistrement article1, il faut utiliser :

```
      article1.ref⟵ "article11_11"
      article1.nom ⟵ "ABC"
      article1.prix ⟵ 200
      article1.fab.ref ⟵ "Fab110 "
      article1.fab.nom ⟵ "Nom_fab_art "
      article1.fab.tel ⟵ "043534 "
```

**Tableau dans une structure**

On peut ajouter un tableau comme champ de structure :

```
Type
Structure nom_type
      .................
      Champ_t : tableau[0..nbelement-1] de type
      .................
FinStructure
```

Pour accéder au tableau, on utilise la syntaxe suivante :

```
nom_enreg.Champ_t[indice]
```

**Exemple :**

Nous voulons connaître le nombre d'articles vendus sur les 12 mois de l'année.

```
Type
  Structure bilanart
    art : tarticle
    vente : tableau[0..11] de réels
  FinStructure
```

Puis, déclarer

```
bart : bilanart
```

```
    Pour i allant de 1 jusqu'à 12 Faire
        Afficher (" Vente du mois ",i, " ?")
        Lire (bart.vente[i-1])
        total ⟵total+bart.vente[i]
    FinPour
```

**Passage d'un enregistrement en paramètre**

Il est possible de passer tout un enregistrement en paramètre d'une fonction ou d'une procédure (on n'est pas obligé de passer tous les champs uns à uns, ce qui permet de diminuer le nombre de paramètres à passer), exactement comme pour les tableaux.

**Exemple :**

```
Procédure Proc_afficher (article1 : tarticle)
    DébutProcédure
        Afficher (article1.ref, article1.nom, article1.prix)
    FinProcédure
```

**Tableaux d'enregistrement (ou tables)**

Il arrive souvent que l'on veuille traiter non pas un seul enregistrement mais plusieurs. Par exemple, on veut pouvoir représenter plusieurs articles. On va créer un tableau regroupant toutes les articles d'un catalogue. Il s'agit alors d'un tableau d'enregistrements.

Soit la structure tarticle :

```
Type
Structure tarticle
    ref : chaîne de caractères
    nom : chaîne de caractères
    prix : Réel
FinStructure
```

Vous voulez créer une table de dix articles :

```
Variable articles : tableau[0..9] de tarticle
```

Dans ce ca, chaque élément du tableau est un enregistrement, contenant plusieurs variables de type différent. On accède à un enregistrement par son indice dans le tableau.

- articles[1] représente le deuxième article du catalogue,
- articles[1].nom représente le nom de deuxième article du catalogue.

Pour accéder aux dix enregistrements, le mieux est dŠutiliser une boucle :

```
    Pour i allant de 1 jusqu'à 10 Faire
        Afficher (" Saisir ref article ",i)
        Lire (articles[i-1].ref)
    FinPour
```

## IV.2 Fichiers

Un fichier est un ensemble structuré de données de même type, nommé et enregistré sur un support lisible par l'ordinateur (disque dur, disquette, flash disque, CD Rom, ..etc). Un fichier peut contenir des caractères (fichier textes), des programmes, des valeurs (fichier de données).

Organisation des fichiers : un fichier se distingue des autres par quelques attributs dont son nom et sa catégorie. Ils se distinguent aussi entre eux par l'organisation de leurs données ce qui définit leur format.

Catégories de fichiers : deux catégories de fichiers sont distinguables :

- Les fichiers organisés sous forme de lignes de texte successives, qui s'appellent des fichiers texte. Cela signifie vraisemblablement que ce fichier contient le même genre d'information à chaque ligne. Ces lignes sont alors appelées des enregistrements.
- Le second type de fichier : il rassemble les fichiers qui ne possèdent pas de structure de lignes (d'enregistrement). Ces fichiers contenant des données variées dont des nombres représentés sous forme binaire. Ces fichiers sont appelés des fichiersăbinaires.

Structure des enregistrements : les fichiers peuvent être structurés en enregistrement. Il y a deux grandes possibilités pour structurer ces enregistrements :

- La structure est dite délimitée ; Elle utilise un caractère spécial, appelé caractère de séparation ou délimitation, qui permet de repérer quand finit un champ et quand commence le suivant.

  Exemple :

  ```
  "Idrissi1" ;"Hassane1" ;0123 ;"Hassane1@yahoo.fr"
  "Idrissi2" ;"Hassane2" ;0456 ;"Hassane2@yahoo.fr"
  "Idrissi3" ;"Hassane3" ;0789 ;"Hassane3@yahoo.fr"
  "Idrissi4" ;"Hassane4" ;0978 ;"Hassane4@yahoo.fr"
  ```

  Remarque : Le caractère de délimitation ne doit pas se retrouver à l'intérieur de chaque champ.
- Une structure est dite à champs de largeur fixe, s'il n'y a pas de délimiteurs. Chaque champ a une longueur prédéfinie et occupe toute cette longueur, quitte à être complété par des espaces.

| | | | |
|---|---|---|---|
| Idrissi1 | Hassane1 | 0123 | Hassane1@yahoo.fr |
| Idrissi2 | Hassane2 | 0456 | Hassane2@yahoo.fr |
| Idrissi3 | Hassane3 | 0789 | Hassane3@yahoo.fr |
| Idrissi4 | Hassane4 | 0978 | Hassane4@yahoo.fr |

**Remarque IV.1** *Contrairement au format limité, le format à largeur fixe consomme*

*bien plus de mémoire. Cependant, la récupération de tels champs est bien plus simple car vous connaissez à l'avance la taille de chaque champ et donc toutes les positions pour découper vos enregistrements.*

### Types d'accès

Le type d'accès est la manière dont la machine va pouvoir aller rechercher les informations contenues dans le fichier. On distingue :

- L'accès séquentiel : Il consiste à traiter les informations "séquentiellement", c'est-à-dire dans l'ordre où elles apparaissent dans le fichier. Donc,
  - Pour lire une information particulière, il faut lire toutes les informations situées avant.
  - On accède à une données quelconque en se déplaçant (via un pointeur) depuis la donnée de départ.
  - On peut ajouter une donnée à la fin du fichier.
- L'accès direct : Il consiste à se placer immédiatement sur l'information souhaitée, sans avoir à parcourir celles qui la précèdent.
  - Nous pouvons donc accéder directement à l'information désirée, en précisant le numéro d'emplacement (le numéro d'ordre) de cette information.
  - On peut modifier n'importe quelle donnée.

### Manipulation les fichiers à accès séquentiel

Pour travailler avec des fichiers, vous devez respecter un certain ordre. Il vous faudra :

- Ouvrir le fichier : c'est-à-dire indiquer à quel fichier vous voulez accéder.
- Traiter le contenu du fichier : le lire, y écrire bref toutes les opérations désirées et manipuler son contenu.
- Fermer le fichier : quand tous les traitements sont terminés.

**Remarque IV.2** *Si l'on veut travailler sur un fichier, la première chose à faire est de l'ouvrir. Cela se fait en attribuant au fichier un numéro de canal. On ne peut ouvrir qu'un seul fichier par canal, mais quel que soit le langage, on dispose toujours de plusieurs canaux. Donc, l'accès à un fichier passe par l'utilisation d'un canal. Un canal permet de faire transiter un flux d'informations d'un programme vers un fichier.*

### Déclaration d'un fichier

La structure fichier se déclare comme un type prédéfini :

| nom_fichier : fichier séquentiel |
|---|

**Remarque IV.3** *La déclaration des fichiers doit figurer avant la déclaration des autres variables.*

### Ouverture d'un fichier

Pour ouvrir un fichier texte, on écrira par exemple :

| Ouvrir NomFichier dans nom_fichier en mode |
|---|

où, NomFichier correspond au nom du fichier concerné (exemple : "Toto.txt" ).

Les modes d'ouverture : l'important est que lorsqu'on ouvre un fichier, on souhaite lire son contenu, y écrire, ou ajouter des lignes à la fin.

- Si on ouvre un fichier pour lecture, on pourra uniquement récupérer les informations qu'il contient, sans les modifier en aucune manière.
- Si on ouvre un fichier pour écriture, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront intégralement écrasées et on ne pourra pas accéder aux informations qui existaient précédemment.
- Si on ouvre un fichier pour ajout, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra, comme vous commencez à vous en douter, ajouter de nouvelles lignes.

**Remarque IV.4** *Il faut toujours fermer un fichier à la fin d'une session. Par exemple :*

| *Fermer nom_fichier* |
|---|

**Exemple :**

```
Algorithme Ouvre
Variable
   fic : fichier séquentiel
   nom : chaîne de caractères
Début
      nom ⟵ "Toto.txt"
      Ouvrir nom dans fic en Lecture
      /* traitements */
      Fermer fic
Fin
```

**Lecture dans un fichier**

Lire des enregistrements (lignes) :
La lecture d'une ligne se fait via l'instruction **Lire**. **Lire** lit l'enregistrement présent à la position actuelle du fichier, puis se place sur l'enregistrement suivant.

La syntaxe est la suivante :

```
Lire(nom_fichier, variable)
```

**Exemple :**

```
Algorithme Lire_fichier
Variable
   fic : fichier séquentiel
   Ligne, Nom, Prénom, Tel, Mail : chaîne de caractères
Début
      Ouvrir "Toto.txt" dans fic en Lecture
      Lire (fic,Ligne)
            Nom ⟵ Milieu(Ligne, 1, 20)
            Prénom ⟵ Milieu(Ligne, 21, 35)
            Tel ⟵ Milieu(Ligne, 36, 45)
            Mail ⟵ Milieu(Ligne, 46, 100)
      Fermer fic
Fin
```

**Remarque IV.5** *La fonction "*Milieu*" : permet d'extraire une chaîne de caractères d'une longueur donnée, à partir d' une position donnée.*

Lire un fichier séquentiel de bout en bout suppose de programmer une boucle. Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, la combine consiste à utiliser la fonction "FinFichier()". Cette fonction prend en paramètre le nom du fichier. Elle retourne Vrai si la fin du fichier a été atteinte.
**Exemple :**

```
Algorithme Lire_fichier
Type
  Structure Personne
    Nom, Prénom, Tel, Mail : chaîne de caractères
  FinStructure
Variable
  fic : fichier séquentiel
  Ligne : chaîne de caractères
  i : Entier
  Base : tableau[ ] de Personne
Début
    Ouvrir "Toto.txt" dans fic en Lecture
    i ⟵ -1
    Tantque (FinFichier(fic)=Faux) Faire
        i ⟵ i+1
        Taille(Base) ⟵ Taille(Base)+1
        Lire (fic,Base[i])
    FinTantque
Fin
```

### Écriture dans un fichier

**Écrire** : l'écriture utilise l'instruction Écrire, une fonction qui prend comme paramètre le nom du fichier et l'enregistrement (la ligne).

La syntaxe est la suivante

Écrire(nom_fichier, ligne)

Avec,

Ouvrir NomFichier dans nom_fichier en Écriture

**Ajouter** : Ajout utilise l'instruction Écrire, une fonction qui prend comme paramètre le nom du fichier et l'enregistrement (la ligne). La syntaxe est la suivante

Écrire(nom_fichier, ligne))

Avec,

Ouvrir NomFichier dans nom_fichier en Ajout

# V La complexité

Quand on tente à résoudre un problème, la question qui se pose c'est le choix du meilleur algorithme parmi les algorithmes qui permettent de décrire des méthodes de résolution de ce problème.

Certains algorithmes sont complexes et le traitement peut nécessiter beaucoup de temps et de ressources de machine, c'est qu'on appelle le "coût" (efficacité ou complexité) de l'algorithme.

Le but de ce chapitre est de donner des outils pour comparer différentes solutions algorithmiques à un problème donné.

L'analyse de la complexité des algorithmes étudie formellement la quantité de ressources en temps et en espace nécessitée par l'exécution d'un algorithme donné.

En général, le temps d'exécution sur machine dépend de

- Le problème à résoudre.
- La taille des données.
- L'algorithme de résolution.
- L'expertise du programmeur.
- L'habilité du programmeur.
- La rapidité de la machine.
- Le langage de programmation.
- Le compilateur.

Dans ce cours nous intéressons au coût des actions résultant de l'exécution d'un algorithme, en fonction de la taille des données traitées. Ceci nous permet de comparer des algorithmes traitant le même problème.

**Définition V.1** *La complexité d'un algorithme désigne le nombre d'opérations fondamentales ( affectation, comparaison, opérations arithmétiques, ...). On la voit comme une fonction de $n$, la taille de l'entrée.*

**Remarque V.1** *Le choix de l'unité de mesure ne dépend pas de la nature précise des données mais de leur taille $n$. Donc, La complexité s'exprime en fonction de la taille $n$ des données.*

Mesure de la complexité : On distingue plusieurs types d'analyses de complexité :

- Le Meilleur des cas (cas favorable).
- Le pire des cas (cas défavorable).
- Cas moyen (complexité moyen pour toutes les entrées possibles).

Par exemple pour la recherche d'un élément dans un tableau :

$$T = [12, 32, 22, 10, 5, 45]$$

- Le meilleur cas est que l'on trouve l'élément à la première comparaison.
- Le pire cas est qu'on parcours tous les éléments de la liste et que l'élément recherché ne s' y trouve pas.
- Le moyen cas est que l'élément recherché se trouve à la 2ème, 3ème position par exemple.

Pour ce cours nous nous intéressons particulièrement au pire des cas.

On désigne par $C(n)$ le nombre d'opérations effectuées pour exécuter un algorithme donné dont la taille de données est $n$.

Il n'est souvent pas pertinent d'essayer de quantifier trop précisément $C(n)$, vu qu'on raisonne au niveau de l'algorithme et non d'une implantation. On se contente donc de classifier $C(n)$ avec un ordre de grandeur en $O$. Donc, on se préoccupe surtout de la croissance de la complexité en fonction de la taille des données.

Plus précisément, $C(n) = O(f(n))$ : signifie que le nombre d'opérations effectuées est borné par $K * f(n)$ lorsque $n$ tend vers l'infini, c'est-à-dire :

Il existe $K > 0$ et $n_0$ telles que pour tout $n \geq n_0$ on a $C(n) \leq Kf(n)$.

Généralement la fonction $f(n)$ est une combinaison de polynômes, logarithmes, ou exponentielle.

## V.1 Classes de complexités classiques

On voit souvent apparaître les complexités suivantes :

- $O(1)$ : complexité constante.
- $O(\log(n))$ (on dit complexité logarithmique) : Ce sont des algorithmes très rapides. Exemples typiques : recherche dichotomique, exponentiation rapide, etc.

- $O(n)$ (on dit complexité linéaire) : Typiquement quand on parcourt un tableau ou une liste un nombre borné de fois : recherche dans un tableau, minimum d'un tableau, etc.
- $O(n * log(n))$ (complexité quasi-linéaire). Exemples : tri rapide, tri fusion, tri par tas, etc. Cette complexité apparaît régulièrement lorsque l'on fait du "diviser pour régner".
- $O(n^2)$ (complexité quadratique). Quand on manipule des tableaux à deux dimensions, ou qu'on effectue un assez grand nombre de calculs sur un tableau à une dimension : somme de deux matrices, transposée d'une matrice, tri insertion, tri bulle, tri selection, etc.
- $O(n^p)$ : complexité polynomiale.
- $O(2^n)$ : complexité exponentielle.
- O(n !) : complexité factorielle. Sur Intel Pentium 4 à 3.2 GHz, si vous traitez 20 données dans un algorithme de complexité $O(n!)$ le temps d'exécution est autour de 25 ans.

## V.2 Calcul de la complexité

- Pour une opération de base comme par exemple affectation, lecture, écriture, comparaison, addition... la complexité dans ce cas est $O(1)$.
- Instruction séquentielle :

$$C(I_1, \cdots, I_p) = \max(C(I_1), \cdots, C(I_p)).$$

- Instructions conditionnelles

$$C(\text{ Si cond alors } I_1 \text{ Sinon } I_2)) = C(\text{cond}) + \max(C(I_1), C(I_2)).$$

- Instruction itératives

$$C(\text{ Pour allant de } i = i_1 \text{ jusqu'à } i_2 \text{ Faire } T_i) = (i_2 - i_1 + 1) \max_{i_1 \leq i \leq i_2} (C(T_i)).$$

-

| | |
|---|---|
| Tant que <Condition> Faire | $C_i(n)$ |
| Bloc d'instructions | $T_i(n)$ |
| Fin Tant que | |

Donc,

$$\sum_{i=1}^{k}(C_i(n) + T_i(n)) + C_{k+1}(n)$$

avec $T_i(n)$ : coût de la $i$ième itération et $k$ le nombre total d'itérations.

Souvent la complexité est définie par une équation récursive.

**Théorème V.1**

- *Si* $C(n+1) = C(n)$ *alors* $C(n) = O(1)$.
- *Si* $C(2n) = C(n) + 1$ *alors* $C(n) = O(\log(n))$.
- *Si* $C(n+1) = C(n) + 1$ *alors* $C(n) = O(n)$.
- *Si* $C(n) = 2 * C(n/2) + n$ *alors* $C(n) = O(nlog(n))$.
- *Si* $C(n+1) = C(n) + n$ *alors* $C(n) = O(n^2)$.
- *Si* $C(n+1) = 2 * C(n)$ *alors* $C(n) = O(2^n)$.

Maintenant, on donne un exemple d'évaluation de $C(n)$ pour les fonctions récursives.

```
Fonction FunctionRecursive (n : Entier) : ....
DébutFonction
    Si (n>1) Alors
      FunctionRecursive(n/2),        coût C(n/2)
      Traitement(n),                 coût T(n)
      FunctionRecursive(n/2),        coût C(n/2)
```

Donc, on a l'équation récursive suivante :

$$C(n) = 2 * C(n/2) + T(n) + 1.$$

Par conséquent,

$$\begin{array}{lll} \text{Si } T(n) = 1 & \text{alors} & C(n) = O(n). \\ \text{Si } T(n) = n & \text{alors} & C(n) = C(n) = O(n \log n). \end{array}$$

## V.3 Exemples importants

Dichotomie :
On veut chercher si un entier $x$ est dans un tableau trié $T$ de taille $n$.

**Théorème V.2**
*La complexité de la fonction Recherche_Dicho est en $O(log(n))$.*

Démonstration :
A Chaque appel, on divise le tableau de recherche en 2. Soit $k$ tel que $n = 2^k$. Donc

$$\begin{aligned} C(n) &= 1 + C(n/2) \\ &= 1 + C(2^{k-1}) \\ &= 2 + C(2^{k-2}) \\ &= ... \\ &= k + C(1) = \log_2(n) + C(1). \end{aligned}$$

Donc, la complexité est en $O(\log_2(n))$.

Exponentiation rapide :
On veut calculer $x^n$, $n \in \mathbb{N}$ mesure la taille de l'entrée. Pour cela on propose deux méthodes récursives.

Méthode 1 :

$$\begin{cases} 1, & \text{si } n = 0, \\ x * x^{n-1}, & \text{si } n > 0. \end{cases}$$

**Théorème V.3**
*La complexité de la méthode 1 est en $O(n)$.*

Démonstration :
Puisque

$$C(n+1) = 1 + C(n).$$

Alors

$$\begin{aligned} C(n+1) &= 1 + C(n) \\ &= 2 + C(n-1) \\ &= ... \\ &= n + C(1). \end{aligned}$$

Donc, la complexité est en $O(n)$.

Méthode 2 :

$$\begin{cases} 1, & \text{si } n = 0, \\ x^k * x^k, & \text{si } n = 2k \text{ et } k > 0. \\ x * x^k * x^k, & \text{si } n = 2k+1. \end{cases}$$

**Théorème V.4**
*La complexité de la méthode* 2 *est en* $O(log(n))$.

Tri rapide :
Maintenant, on étudie la complexité de la fonction récursive "TriRapide", qui est basée sur le paradigme diviser pour régner, pour trier un tableau $T$ de taille $n$. Supposons que le pivot, après le partitionnement, se trouve à la position $p$, $0 \leq p \leq n-1$, c'est-à-dire $pivot = T[p]$. La formule de récurrence suivante somme respectivement le coût du calcul du partitionnement et du test, soit $O(1) + O(n) = O(n)$ et des deux appels récursifs :

$$C(n) = O(n) + C(p+1) + C(n-p-1)$$

avec $C(1) = O(1)$.

Dans le cas idéal, la partition constitue deux parties de tailles égales, c'est-à-dire $p = n/2 - 1$. Donc

$$C(n) = O(n) + 2C(n/2) = O(n * \log_2(n)).$$

En effet, posons $n = 2^k$

$$\begin{aligned}
C(n) &= n + 2 * C(n/2) \\
&= 2^k + 2 * C(2^{k-1}) \\
&= 2^k + 2 * (2^{k-1} + 2 * C(2^{k-2})) = 2^k + 2^k + 2^2 * C(2^{k-2}) \\
&= ... \\
&= k * 2^k + 2^k * C(1) = O(n * \log_2(n)).
\end{aligned}$$

Dans le pire des cas, c'est à dire quand une des parties est vide et l'autre contient $n - 1$ éléments. Dans ce cas, on a $p = 0$. Par conséquent,

$$C(n) = O(n) + C(1) + C(n-1) = O(n^2).$$

En effet,

$$\begin{aligned}
C(n) &= n + C(1) + C(n-1) \\
&= n + (n-1) + 2C(1) + C(n-2) \\
&= ... \\
&= O(n^2).
\end{aligned}$$

# VI Preuves d'algorithmes

## VI.1 Propriétés recherchées

Une preuve algorithmique est en général assez délicate. Souvent nos algorithmes sont très simples, et la preuve ressemble à une paraphrase de l'algorithme. Malheureusement, il n'existe pas d'outil automatique pour vérifier la correction d'algorithmes.

La correction d'un algorithme s'étudie par rapport à un problème donné. Un problème est une collection d'instances de ce problème.

- Exemple de problème : Trier un tableau.
- Exemple d'instance de ce problème : trier le tableau [8,4,15,3].

**Définition VI.1**

- *Un algorithme est correct pour une instance d'un problème s'il produit une solution correcte pour cette instance.*
- *Un algorithme est correct pour un problème s'il est correct pour toutes ses instances (on dira qu'il est totalement correct).*

On s'intéressera ici à la correction d'un algorithme pour un problème (et pas pour seulement certaines de ses instances).

La question qui se pose maintenant comment vérifier la correction d'un algorithme. En général, il y a deux solutions.

1. Première solution : en testant l'algorithme :
   - Supposons d'implémenter l'algorithme dans un langage et de le faire tourner.
   - Supposons qu'on peut déterminer les instances du problème à vérifier.
   - Il est très difficile de prouver empiriquement qu'on n'a pas de bug.
2. Deuxième solution : en dérivant une preuve mathématique.

Il existe des outils mathématiques pour prouver la correction d'un algorithme :

- Algorithmes itératifs : triplets de Hoare, invariants de boucle.
- Algorithmes récursifs : preuves par induction.

En informatique, prouver un algorithme ça signifie deux choses :

1. Prouver sa terminaison : un algorithme doit effectuer un nombre fini d'étapes et s'arrêter.
2. Prouver sa correction : un algorithme doit faire ce qu'on attend de lui.

## VI.2 Terminaison d'un algorithme

**Définition VI.2** *Terminaison :*
*On dit qu'un algorithme $\mathcal{P}$ termine, si et seulement si, tout état initial $E$ donne une exécution terminante de $\mathcal{P}$.*

L'algorithme suivant ne termine pas :

```
Tant que <condition> Faire



Fintantque
```

L'algorithme suivant ne termine pas :

```
Si (a<=b) Alors
max ⟵ b
Sinon
  Tant que (a <> b) faire
    max ⟵ a
  Fintantque
FinSi
```

## VI.3 Correction d'un algorithme

**Définition VI.3** *(Correction)*
*Soit l'algorithme $A$ dont la correction est exprimée par une condition $C$, On dit que $A$ est correct si et seulement si, pour tout état initial $E$ qui donne une exécution terminante*

$$F = A(E) \text{ on a } F(C) = Vrai.$$

## VI.4 Triplet de Hoare

**Définition VI.4** *(Assertion)*
*Relation entre les variables qui vraie à un moment dans l'exécution.*
*Il existe des assertions particulières :*

- *Pré-condition $P$ : conditions qui doivent remplir les entrées valides de l'algorithme.*
- *Post-condition $Q$ : conditions qui expriment que le résultat de l'algorithme est correct.*

*$P$ et $Q$ définissent resp. les instances et solutions valides du problème.*

**Définition VI.5** *(Triplet de Hoare) Un code est correct si le triplet (de Hoare) :*

$$\{P\} \text{ code } \{Q\} \text{ est vrai}$$
$$\{\text{hypothèses de départ }\} \text{ code } \{\text{résultat}\} \text{ est vrai}$$

Exemple :

$$\{x = 3\}\ x \longleftarrow x + 2\ \ \{x = 5\}$$

Si $x$ vaut 3 avant l'instruction $x \longleftarrow x + 2$ alors après l'instruction $x$ vaut 5.
Prenons le cas de l'affectation suivante

$$\{P\}\ x \longleftarrow e\ \{Q\}$$

Pour prouver que le triplet est correct, il faut montrer qu'on peut dériver $Q$ à partir de $P$ en remplaçant $x$ par $e$.

Exemple :

$$\{x = 2\}\ y \longleftarrow x + 1\ \ \{y = 3\}$$

Correction des Conditions. Soit

| |
|---|
| $\{P\}$ |
| Si $B$ Alors |
| $C_1$ |
| Sinon |
| $C_2$ |
| FinSi |
| $\{Q\}$ |

Pour prouver que le triplet est correct on doit prouver que

- $\{P$ et $B\}$ $C_1$ $\{Q\}$
- $\{P$ et non $B\}$ $C_2$ $\{Q\}$

sont corrects.

Exemple :

$\{x < 6\}$
Si $x < 0$ Alors
$y \longleftarrow 0$
Sinon
$y \longleftarrow x$
FinSi
$\{0 \leq y \leq 6\}$

Correction des boucles. Soit

$\{P\}$ INIT $\{I\}$
Tant que $B$ faire
$\{I$ et $B\}$ Corps $\{I\}$
$\{I$ et non $B\}$
Fintantque
$\{Q\}$

Pour prouver le triplet est correct :

- On met en évidence une assertion particulière $I$, appelée invariant de boucle, qui décrit l'état du programme pendant la boucle.
- on prouve que
  1. $\{P\}$ INIT $\{I\}$ est correct.
  2. $\{I$ et $B\}$ Corps $\{I\}$ est correct.
  3. $\{I$ et non $B\}$ Fin Tant que $\{Q\}$ est correct.

## VI.5 Preuve par invariant de boucle

Une preuve d'algorithme par invariant de boucle utilise la démarche suivante. Nous prouvons tout d'abord que l'algorithme s'arrête en montrant qu'une condition d'exécution de boucle finit par ne plus être réalisée. Nous exhibons alors un

invariant de boucle, c'est-à-dire une propriété $P$ qui, si elle est valide avant l'exécution d'un tour de boucle, est aussi valide après l'exécution du tour de boucle. Nous vérifions alors que les conditions initiales rendent la propriété $P$ vraie en entrée du premier tour de boucle. Nous en concluons que cette propriété est vraie en sortie du dernier tour de boucle. Un bon choix de la propriété $P$ prouvera qu'on a bien produit l'objet recherché. La difficulté de cette méthode réside dans la détermination de l'invariant de boucle. Quand on l'a trouvé il est en général simple de montrer que c'est bien un invariant de boucle.

Montrer que l'algorithme suivant se termine et en sortie R contient $x^n$.

```
A ⟵ x
N ⟵ n
R ⟵ 1
Tant que (N > 0) faire
  Si (N mod 2=0) Alors
    A ⟵ A*A
    N ⟵ N/2
  Sinon
    R ⟵ R*A
    N ⟵ N-1
  FinSi
Fintantque
```

**Preuve :** La valeur de $N \geq O$ décroît strictement à chaque tour de boucle, donc l'algorithme se termine.
Au début on a $A^N \times R = x^n$. Si en entrée de boucle $A^N \times R = x^n$ alors il est facile de voir que dans les deux cas $N$ pair et $N$ impair on a la même égalité en sortie de boucle. Mais à la fin on a $N = 0$ et par conséquent $R = x^n$.

## VI.6 Correction d'une fonctions récursive

**Définition VI.6** *Invariant :*
*Une condition vérifiée sur l'état des données traitées par la récurrence.*

Respect Invariant : l'invariant doit être vérifié par chaque étape (appel) de la résolution récursive.

Principe de récurrence : pour prouver ce que fait un programme récursif, on fait en général appel au principe de récurrence.

Étant donnée une propriété $P$ sur les entiers, la formule suivante est vérifiée : $[P(0)$ et pour tout $n$, $P(n)$ implique $P(n+1)]$ Alors pour tout $n$, $P(n)$.